6

Linear-Time Construction of Suffix Trees

We will present two methods for constructing suffix trees in detail, Ukkonen's method and Weiner's method. Weiner was the first to show that suffix trees can be built in linear time, and his method is presented both for its historical importance and for some different technical ideas that it contains. However, Ukkonen's method is equally fast and uses far less space (i.e., memory) in practice than Weiner's method. Hence Ukkonen is the method of choice for most problems requiring the construction of a suffix tree. We also believe that Ukkonen's method is easier to understand. Therefore, it will be presented first. A reader who wishes to study only one method is advised to concentrate on it. However, our development of Weiner's method does not depend on understanding Ukkonen's algorithm, and the two algorithms can be read independently (with one small shared section noted in the description of Weiner's method).

6.1. Ukkonen's linear-time suffix tree algorithm

Esko Ukkonen [438] devised a linear-time algorithm for constructing a suffix tree that may be the conceptually easiest linear-time construction algorithm. This algorithm has a space-saving improvement over Weiner's algorithm (which was achieved first in the development of McCreight's algorithm), and it has a certain "on-line" property that may be useful in some situations. We will describe that on-line property but emphasize that the main virtue of Ukkonen's algorithm is the simplicity of its description, proof, and time analysis. The simplicity comes because the algorithm can be developed as a simple but inefficient method, followed by "common-sense" implementation tricks that establish a better worst-case running time. We believe that this less direct exposition is more understandable, as each step is simple to grasp.

6.1.1. Implicit suffix trees

Ukkonen's algorithm constructs a sequence of *implicit* suffix trees, the last of which is converted to a true suffix tree of the string S.

Definition An *implicit suffix tree* for string S is a tree obtained from the suffix tree for S by removing every copy of the terminal symbol $ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children.

An implicit suffix tree for a prefix $S[1..i]$ of S is similarly defined by taking the suffix tree for $S[1..i]$$ and deleting $ symbols, edges, and nodes as above.

Definition We denote the implicit suffix tree of the string $S[1..i]$ by $\mathcal{I}_i$, for i from 1 tom.

The implicit suffix tree for any string S will have fewer leaves than the suffix tree for

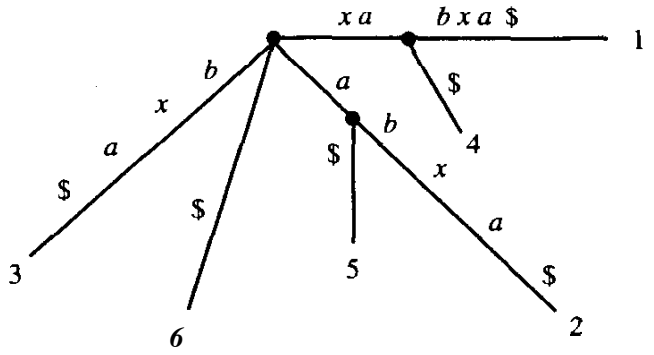

Figure 6.1: Suffix tree for string ***xabxa*** $.

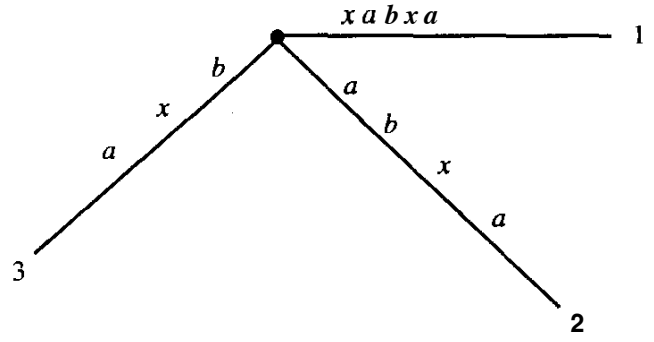

Figure 6.2: Implicit suffix tree for string ***xabxa.***

string S\$ if and only if at least one of the suffixes of S is a prefix of another suffix. The terminal symbol \$ was added to the end of S precisely to avoid this situation. However, if S ends with a character that appears nowhere else in S, then the implicit suffix tree of S will have a leaf for each **suffix** and will hence be a true suffix tree.

As an example, consider the suffix tree for string ***xabxa***\$ shown in Figure 6.1. Suffix ***xa*** is a prefix of suffix ***xabxa,*** and similarly the string ***a*** is a prefix of ***abxa.*** Therefore, in the suffix tree for ***xabxa*** the edges leading to leaves 4 and 5 are labeled only with \$. Removing these edges creates two nodes with only one child each, and these are then removed as well. The resulting implicit suffix tree for ***xabxa*** is shown in Figure 6.2. As another example, Figure 5.1 on page 91 shows a tree built for the string ***xabxac.*** Since character ***c*** appears only at the end of the string, the tree in that figure is both a suffix tree and an implicit suffix tree for the string.

Even though an implicit suffix tree may not have a leaf for each suffix, it does encode all the suffixes of S – each suffix is spelled out by the characters on some path from the root of the implicit suffix tree. However, if the path does not end at a leaf, there will be no marker to indicate the path's end. Thus implicit suffix trees, on their own, are somewhat less informative than true suffix trees. We will use them just as a tool in Ukkonen's algorithm to finally obtain the true suffix tree for S.

6.1.2. Ukkonen's algorithm at a high level

Ukkonen's algorithm constructs an implicit suffix tree $\mathcal{I}_i$ for each prefix S[1..i] of S, starting from $\mathcal{I}_1$ and incrementing i by one until $\mathcal{I}_m$ is built. The true suffix tree for S is constructed from $\mathcal{I}_m$, and the time for the entire algorithm is $O(m)$. We will explain

Ukkonen's algorithm by first presenting an $0(m^3)$-time method to build all trees $\mathcal{T}_i$ and then optimizing its implementation to obtain the claimed time bound.

High-level description of Ukkonen's algorithm

Ukkonen's algorithm is divided into m phases. In phase i + 1, tree $\mathcal{T}_{i+1}$ is constructed from $\mathcal{T}_i$. Each phase i + 1 is further divided into i + 1 extensions, one for each of the $i + 1$ suffixes of $S[1..i + 1]$. In extension j of phase i + 1, the algorithm first finds the end of the path from the root labeled with substring $S[j..i]$. It then extends the substring by adding the character S(i + 1) to its end, unless S(i + 1) already appears there. So in phase $i + 1$, string $S[1..i + 1]$ is first put in the tree, followed by strings $S[2..i + 1]$, $S[3..i + 1]$, ... (in extensions 1,2, 3, ..., respectively). Extension i + 1 of phase i + 1 extends the empty suffix of $S[1..i]$, that is, it puts the single character string S(i + 1) into the tree (unless it is already there). Tree $\mathcal{T}_1$ is just the single edge labeled by character $S(1)$. Procedurally, the algorithm is as follows:

High-level Ukkonen algorithm

```
Construct tree T1.
For i from 1 to m – 1 do
begin {phase i + 1}
    For j from 1 to i + 1
        begin {extension j}
        Find the end of the path from the root labeled S[j..i] in the
        current tree. If needed, extend that path by adding character S(i + 1),
        thus assuring that string S[j..i + 1] is in the tree.
        end;
end;
```

Suffix extension rules

To turn this high-level description into an algorithm, we must specify exactly how to perform a *suffix* extension. Let $S[j..i] = \beta$ be a suffix of $S[1..i]$. In extension j, when the algorithm finds the end of β in the current tree, it extends β to be sure the suffix βS(i + 1) is in the tree. It does this according to one of the following three rules:

Rule 1 In the current tree, path β ends at a leaf. That is, the path from the root labeled β extends to the end of some leaf edge. To update the tree, character S(i + 1) is added to the end of the label on that leaf edge.

Rule 2 No path from the end of string β starts with character S(i + 1), but at least one labeled path continues from the end of β.

In this case, a new leaf edge starting from the end of β must be created and labeled with character S(i + 1). A new node will also have to be created there if β ends inside an edge. The leaf at the end of the new leaf edge is given the number j.

Rule 3 Some path from the end of string β starts with character S(i + 1). In this case the string $\beta S(i + 1)$ is already in the current tree, so (remembering that in an implicit suffix tree the end of a suffix need not be explicitly marked) we do nothing.

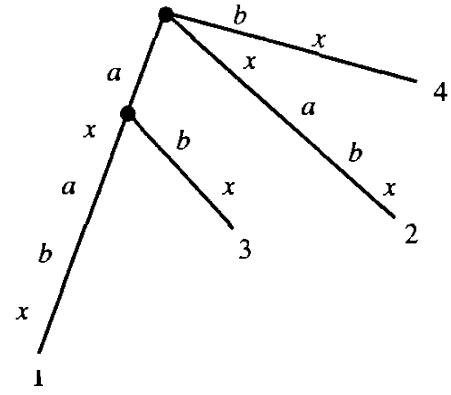

Figure 6.3: Implicit suffix tree for string *axabx* before the sixth character, *b*, is added.

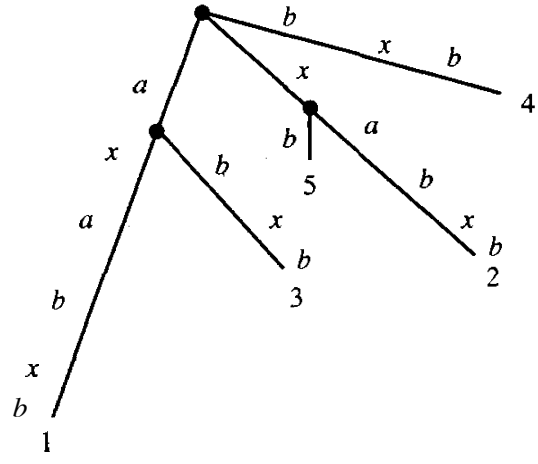

Figure 6.4: Extended implicit suffix tree after the addition of character *b*.

As an example, consider the implicit suffix tree for S = *axabx* shown in Figure 6.3. The first four suffixes end at leaves, but the single character suffix *x* ends inside an edge. When a sixth character *b* is added to the string, the first four suffixes get extended by applications of Rule 1, the fifth suffix gets extended by rule 2, and the sixth by rule 3. The result is shown in Figure 6.4.

6.1.3. Implementation and speedup

Using the suffix extension rules given above, once the end of a suffix β of $S[1..i]$ has been found in the current tree, only constant time is needed to execute the extension rules (to ensure that suffix $\beta S(i+1)$ is in the tree). The key issue in implementing Ukkonen's algorithm then is how to locate the ends of all the $i+1$ suffixes of $S[1..i]$.

Naively we could find the end of any suffix β in $O(|\beta|)$ time by walking from the root of the current tree. By that approach, extension j of phase i + 1 would take O(i + 1 − j) time, $\mathcal{I}_{i+1}$ could be created from $\mathcal{I}_i$ in $O(i^2)$ time, and $\mathcal{I}_m$ could be created in $0(m^3)$ time. This algorithm may seem rather foolish since we already know a straightforward algorithm to build a suffix tree in $0(m^2)$ time (and another is discussed in the exercises), but it is easier to describe Ukkonen's $O(m)$ algorithm as a speedup of the $0(m^3)$ method above.

We will reduce the above $0(m^3)$ time bound to $O(m)$ time with a few observations and implementation tricks. Each trick by itself looks like a sensible heuristic to accelerate the algorithm, but acting individually these tricks do not necessarily reduce the worst-case

bound. However, taken together, they do achieve a linear worst-case time. The most important element of the acceleration is the use of ***suffix links.***

Suffix links: first implementation speedup

Definition Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly empty) substring. For an internal node v with path-label $x\alpha$, *if* there is another node $s(v)$ with path-label a , then a pointer from v to $s(v)$ is called a ***suffix link.***

We will sometimes refer to a suffix link from v to $s(v)$ as the pair $(v, s(v))$. For example, in Figure 6.1 (on page 95) let v be the node with path-label xa and let $s(v)$ be the node whose path-label is the single character a. Then there exists a suffix link from node v to node $s(v)$. In this case, a is just a single character long.

As a special case, if a is empty, then the suffix link from an internal node with path-label xa goes to the root node. The root node itself is not considered internal and has no suffix link from it.

Although definition of suffix links does not imply that every internal node of an implicit suffix tree has a suffix link from it, it will, in fact, have one. We actually establish something stronger in the following lemmas and corollaries.

Lemma 6.1.1. *If a new internal node v with path-label* $x\alpha$ is *added to the current tree in extension j of some phase* $i + 1$, *then either the path labeled a already ends at an internal node of the current tree or an internal node at the end of string a will be created (by the extension rules) in extension* $j + 1$ *in the same phase* $i + 1$.

PROOF A new internal node v is created in extension j (of phase $i + 1$) only when extension rule 2 applies. That means that in extension j, the path labeled $x\alpha$ continued with some character other than S(i + 1), say c. Thus, in extension j + 1, there is a path labeled α in the tree and it certainly has a continuation with character c (although possibly with other characters as well). There are then two cases to consider: Either the path labeled a continues only with character c or it continues with some additional character. When a is continued only by c, extension rule 2 will create a node $s(v)$ at the end of path a. When a is continued with two different characters, then there must already be a node $s(v)$ at the end of path a. The Lemma is proved in either case. □

Corollary 6.1.1. In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

PROOF The proof is inductive and is true for tree $\mathcal{I}_1$ since $\mathcal{I}_1$ contains no internal nodes. Suppose the claim is true through the end of phase i, and consider a single phase i + 1. By Lemma 6.1.1, when a new node v is created in extension j, the correct node $s(v)$ ending the suffix link from v will be found or created in extension $j + 1$. No new internal node gets created in the last extension of a phase (the extension handling the single character suffix S(i + 1)), so all suffix links from internal nodes created in phase $i + 1$ are known by the end of the phase and tree $\mathcal{I}_{i+1}$ has all its suffix links. □

Corollary 6.1.1 is similar to Theorem 6.2.5, which will be discussed during the treatment of Weiner's algorithm, and states an important fact about implicit suffix trees and ultimately about suffix trees. For emphasis, we restate the corollary in slightly different language.

Corollary 6.1.2. In any implicit suffix tree $\mathcal{I}_i$, if internal node v has path-label $x\alpha$, then *there* is a node $s(v)$ of $\mathcal{I}_i$ with path-label α.

Following Corollary 6.1.1, all internal nodes in the changing tree will have suffix links from them, except for the most recently added internal node, which will receive its suffix link by the end of the next extension. We now show how suffix links are used to speed up the implementation.

Following a trail of suffix links to build $\mathcal{T}_{i+1}$

Recall that in phase i + 1 the algorithm locates suffix $S[j..i]$ of $S[1..i]$ in extension j, for j increasing from 1 to i + 1. Naively, this is accomplished by matching the string $S[j..i]$ along a path from the root in the current tree. Suffix links can shortcut this walk and each extension. The first two extensions (for $j = 1$ and $j = 2$) in any phase i + 1 are the easiest to describe.

The end of the full string $S[1..i]$ must end at a leaf of $\mathcal{T}_i$ since $S[1..i]$ is the longest string represented in that tree. That makes it easy to find the end of that suffix (as the trees are constructed, we can keep a pointer to the leaf corresponding to the current full string $S[1..i]$), and its suffix extension is handled by Rule 1 of the extension rules. So the first extension of any phase is special and only takes constant time since the algorithm has a pointer to the end of the current full string.

Let string $S[1..i]$ be xa, where x is a single character and a is a (possibly empty) substring, and let $(v, 1)$ be the tree-edge that enters leaf 1. The algorithm next must find the end of string $S[2..i] = a$ in the current tree derived from $\mathcal{T}_i$. The key is that node v is either the root or it is an interior node of $\mathcal{T}_i$. If it is the root, then to find the end of α the algorithm just walks down the tree following the path labeled a as in the naive algorithm. But if v is an internal node, then by Corollary 6.1.2 (since v was in $\mathcal{T}_i$) v has a suffix link out of it to node $s(v)$. Further, since $s(v)$ has a path-label that is a prefix of string a, the end of string a must end in the subtree of $s(v)$. Consequently, in searching for the end of a in the current tree, the algorithm need not walk down the entire path from the root, but can instead begin the walk from node $s(v)$. That is the main point of including suffix links in the algorithm.

To describe the second extension in more detail, let y denote the edge-label on edge $(v, 1)$. To find the end of ay, walk up from leaf 1 to node v; follow the suffix link from v to $s(v)$; and walk from $s(v)$ down the path (which may be more than a single edge) labeled y. The end of that path is the end of α (see Figure 6.5). At the end of path a, the tree is updated following the suffix extension rules. This completely describes the first two extensions of phase i + 1.

To extend any string $S[j..i]$ to $S[j..i+1]$ for $j > 2$, repeat the same general idea: Starting at the end of string $S[j-1..i]$ in the current tree, walk up at most one node to either the root or to a node v that has a suffix link from it; let y be the edge-label of that edge; assuming v is not the root, traverse the suffix link from v to $s(v)$; then walk down the tree from $s(v)$, following a path labeled y to the end of $S[j..i]$; finally, extend the suffix to $S[j..i+1]$ according to the extension rules.

There is one minor difference between extensions for $j > 2$ and the first two extensions. In general, the end of $S[j-1..i]$ may be at a node that itself has a suffix link from it, in which case the algorithm traverses that suffix link. Note that even when extension rule 2 applies in extension $j - 1$ (so that the end of $S[j-1..i]$ is at a newly created internal node w), if the parent of w is not the root, then the parent of w already has a suffix link out of it, as guaranteed by Lemma 6.1.1. Thus in extension j the algorithm never walks up more than one edge.

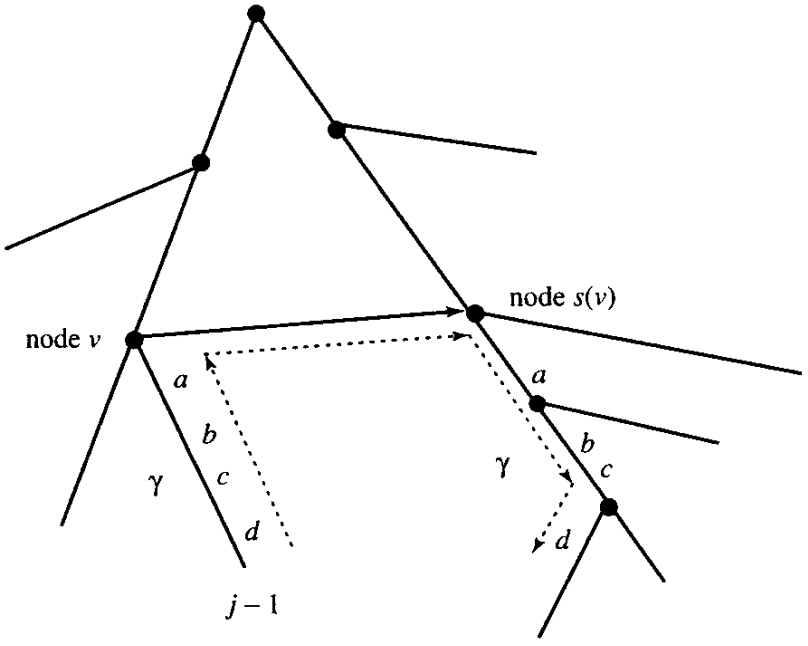

Figure 6.5: Extension $j > 1$ in phase $i+1$. Walk up atmost one edge (labeled γ) from the end of the path labeled $S[j-1..i]$ to node v; then follow the suffix link to $s(v)$; then walk down the path specifying substring γ; then apply the appropriate extension rule to insert suffix $S[j..i+1]$.

Single extension algorithm: SEA

Putting these pieces together, when implemented using suffix links, extension $j \geq 2$ of phase $i + 1$ is:

Single extension algorithm

Begin

1. Find the first node v at or above the end of $S[j-1..i]$ that either has a suffix link from it or is the root. This requires walking up at most one edge from the end of $S[j-1..i]$ in the current tree. Let y (possibly empty) denote the string between v and the end of $S[j-1..i]$.
2. If v is not the root, traverse the suffix link from v to node $s(v)$ and then walk down from $s(v)$ following the path for string γ. If v is the root, then follow the path for $S[j..i]$ from the root (as in the naive algorithm).
3. Using the extension rules, ensure that the string $S[j..i]S(i+1)$ is in the tree.
4. If a new internal node w was created in extension $j-1$ (by extension rule 2), then by Lemma 6.1.1, string α must end at node $s(w)$, the end node for the suffix link from w. Create the suffix link $(w, s(w))$ from w to $s(w)$.

End.

Assuming the algorithm keeps a pointer to the current full string $S[1..i]$, the first extension of phase i + 1 need not do any up or down walking. Furthermore, the first extension of phase i + 1 always applies **suffix** extension rule 1.

What has been achieved so far?

The use of suffix links is clearly a practical improvement over walking from the root in each extension, as done in the naive algorithm. But does their use improve the worst-case running time?

The answer is that as described, the use of suffix links does not yet improve the time bound. However, here we introduce a trick that will reduce the worst-case time for the

algorithm to $O(m^2)$. This trick will also be central in other algorithms to build and use suffix trees.

Trick number 1: skip/count trick

In Step 2 of extension $j+1$ the algorithm walks *down* from node $s(v)$ along a path labeled y. Recall that there surely must be such a y path from $s(v)$. Directly implemented, this walk along γ takes time proportional to $|y|$, the number of *characters* on that path. But a simple trick, called the *skip/count trick,* will reduce the traversal time to something proportional to the *number* of *nodes* on the path. It will then follow that the time for all the down walks in a phase is at most $O(m)$.

Trick 1 Let g denote the length of y, and recall that no two labels of edges out of $s(v)$ can start with the same character, so the first character of γ must appear as the first character on exactly one edge out of $s(v)$. Let g' denote the number of characters on that edge. If g' is less than g, then the algorithm does not need to look at any more of the characters on that edge; it simply skips to the node at the end of the edge. There it sets g to $g - g'$, sets a variable h to $g' + 1$, and looks over the outgoing edges to find the correct next edge (whose first character matches character h of y). In general, when the algorithm identifies the next edge on the path it compares the current value of g to the number of characters g' on that edge. When g is at least as large as g', the algorithm skips to the node at the end of the edge, sets g to $g - g'$, sets h to $h + g'$, and finds the edge whose first character is character h of γ and repeats. When an edge is reached where g is smaller than or equal to g', then the algorithm skips to character g on the edge and quits, assured that the γ path from $s(v)$ ends on that edge exactly g characters down its label. (See Figure 6.6).

Assuming simple and obvious implementation details (such as knowing the number of characters on each edge, and being able, in constant time, to extract from S the character at any given position) the effect of using the skip/count trick is to move from one node to the next node on the y path in *constant* time.[1] The total time to traverse the path is then proportional to the number of *nodes* on it rather than the number of characters on it.

This is a useful heuristic, but what does it buy in terms of worst-case bounds? The next lemma leads immediately to the answer.

Definition Define the *node-depth* of a node u to be the number of *nodes* on the path from the root to u.

Lemma 6.1.2. *Let* $(v, s(v))$ *be any suffix link traversed during Ukkonen's algorithm. At that moment, the node-depth of* v *is at most one greater than the node depth of* $s(v)$.

PROOF When edge $(v, s(v))$ is traversed, any internal ancestor of v, which has path-label $x\beta$ say, has a suffix link to a node with path-label β. But $x\beta$ is a prefix of the path to v, so β is a prefix of the path to $s(v)$ and it follows that the suffix link from any internal ancestor of v goes to an ancestor of $s(v)$. Moreover, if β is nonempty then the node labeled by β is an internal node. And, because the node-depths of any two ancestors of v must differ, each ancestor of v has a suffix link to a distinct ancestor of $s(v)$. It follows that the node-depth of $s(v)$ is at least one (for the root) plus the number of internal ancestors of v who have path-labels more than one character long. The only extra ancestor that v can have (without a corresponding ancestor for $s(v)$) is an internal ancestor whose path-label

[1] Again, we are assuming a constant-sized alphabet.

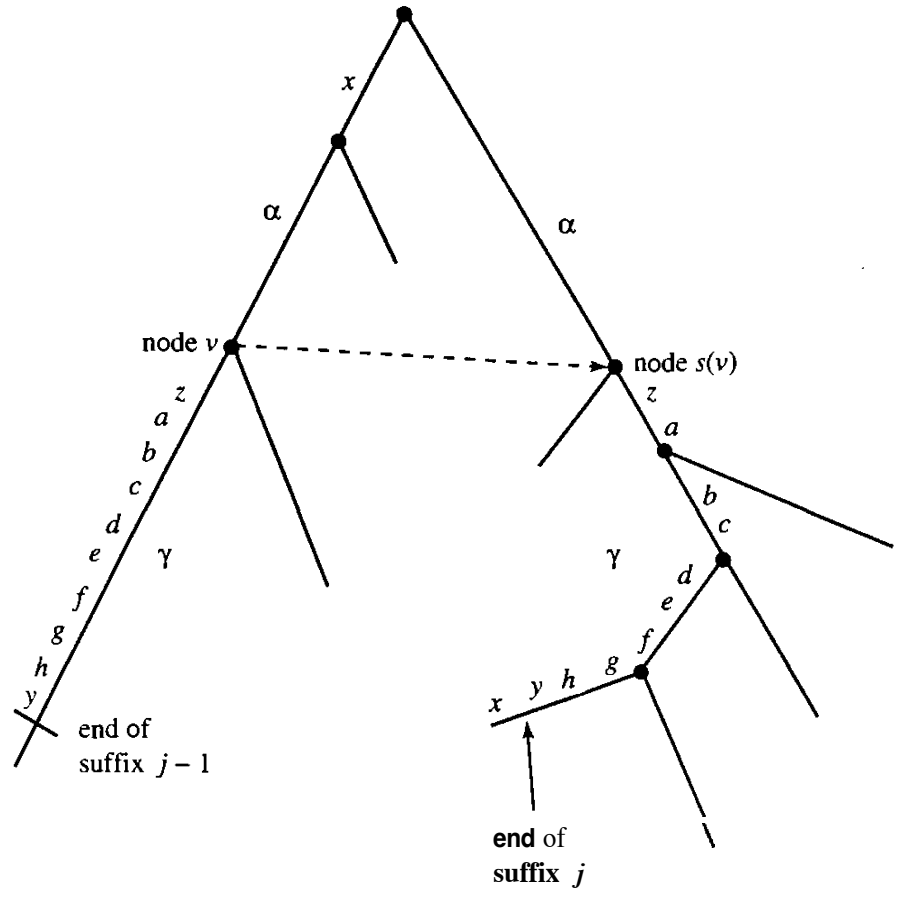

Figure 6.6: The skip/count trick. In phase i + 1, substring γ has length ten. There is a copy of substring y out of node $s(v)$; it is found three characters down the last edge, after four node skips are executed.

has length one (it has label x). Therefore, v can have node-depth at most one more than s(v). (See Figure 6.7). □

Definition As the algorithm proceeds, the ***current node-depth*** of the algorithm is the node depth of the node most recently visited by the algorithm.

Theorem 6.1.1. *Using the skip/count trick, any phase of Ukkonen's algorithm takes $O(m)$ time.*

PROOF There are i + 1 $\leq$ m extensions in phase i. In a single extension the algorithm walks up at most one edge to find a node with a suffix link, traverses one suffix link, walks down some number of nodes, applies the suffix extension rules, and maybe adds a suffix link. We have already established that all the operations other than the down-walking take constant time per extension, so we only need to analyze the time for the down-walks. We do this by examining how the current node-depth can change over the phase.

The up-walk in any extension decreases the current node-depth by at most one (since it moves up at most one node), each suffix link traversal decreases the node-depth by at most another one (by Lemma 6.1.2), and each edge traversed in a down-walk moves to a node of greater node-depth. Thus over the entire phase the current node-depth is decremented at most 2m times, and since no node can have depth greater than m, the total possible increment to current node-depth is bounded by 3m over the entire phase. It follows that over the entire phase, the total number of edge traversals during down-walks is bounded by 3m. Using the skip/count trick, the time per down-edge traversal is constant, so the total time in a phase for all the down-walking is $O(m)$, and the theorem is proved. □

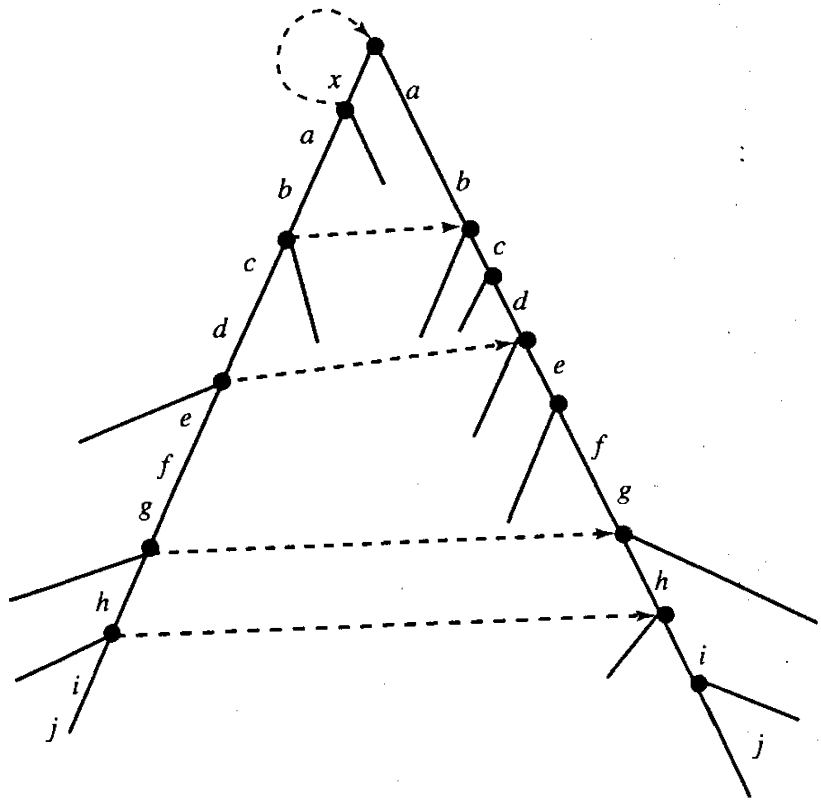

Figure 6.7: For every node v on the path $x\alpha$, the corresponding node $s(v)$ is on the path α. However, the node-depth of $s(v)$ can be one less than the node-depth of v, it can be equal, or it can be greater. For example, the node labeled xab has node-depth two, whereas the node-depth of *ab* is one. The node-depth of the node labeled *xabcdefg* is four, whereas the node-depth of *abcdefg* is five.

There are m phases, so the following is immediate:

Corollary 6.1.3. Ukkonen's algorithm can be implemented with suffix links to run in $O(m^2)$ time.

Note that the $O(m^2)$ time bound for the algorithm was obtained by multiplying the $O(m)$ time bound on a single phase by m (since there are m phases). This crude multiplication was necessary because the time analysis was directed to only a single phase. What is needed are some changes to the implementation allowing a time analysis that crosses phase boundaries. That will be done shortly.

At this point the reader may be a bit weary because we seem to have made no progress, since we started with a naive $O(m^2)$ method. Why all the work just to come back to the same time bound? The answer is that although we have made no progress on the time bound, we have made great conceptual progress so that with only a few more easy details, the time will fall to $O(m)$. In particular, we will need one simple implementation detail and two more little tricks.

6.1.4. A simple implementation detail

We next establish an $O(m)$ time bound for building a suffix tree. There is, however, one immediate barrier to that goal: The suffix tree may require $\Theta(m^2)$ space. As described so far, the edge-labels of a suffix tree might contain more than $\Theta(m)$ characters in total. Since the time for the algorithm is at least as large as the size of its output, that many characters makes an $O(m)$ time bound impossible. Consider the string $S = abcdefghijklmnopqrstuvwxyz$. Every suffix begins with a distinct character; hence there are 26 edges out of the root and

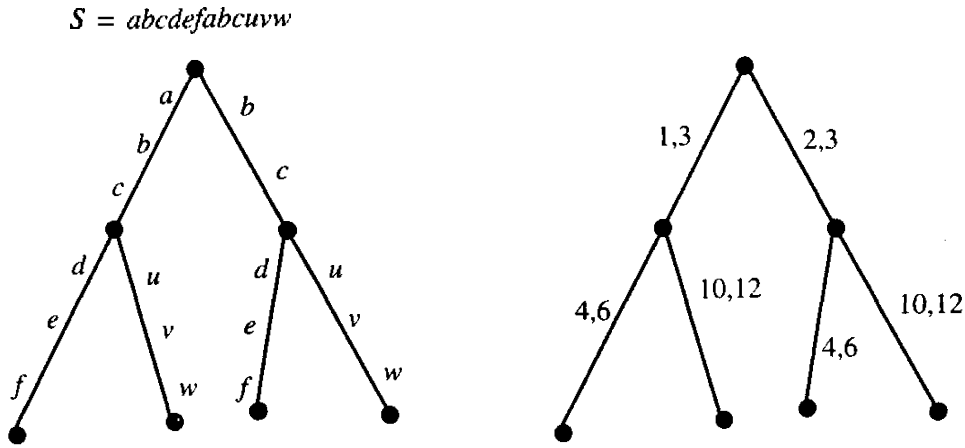

Figure 6.8: The left tree is a fragment of the suffix tree for string S = *abcdefabcuvw,* with the edge-labels written explicitly. The right tree shows the edge-labels compressed. Note that that edge with label 2, 3 could also have been labeled 8, 9.

each is labeled with a complete suffix, requiring $26 \times 27/2$ characters in all. For strings longer than the alphabet size, some characters will repeat, but still one can construct strings of arbitrary length m so that the resulting edge-labels have more than $\Theta(m)$ characters in total. Thus, an $O(m)$-time algorithm for building suffix trees requires some alternate scheme to represent the edge-labels.

Edge-label compression

A simple, alternate scheme exists for edge labeling. Instead of explicitly writing a substring on an edge of the tree, only write a pair of *indices* on the edge, specifying beginning and end positions of that substring in S (see Figure 6.8). Since the algorithm has a copy of string S, it can locate any particular character in S in constant time given its position in the string. Therefore, we may describe any particular suffix tree algorithm as if edge-labels were explicit, and yet implement that algorithm with only a constant number of symbols written on any edge (the index pair indicating the beginning and ending positions of a substring).

For example, in Ukkonen's algorithm when matching along an edge, the algorithm uses the index pair written on an edge to retrieve the needed characters from S and then performs the comparisons on those characters. The extension rules are also easily implemented with this labeling scheme. When extension rule 2 applies in a phase $i+1$, label the newly created edge with the index pair $(i+1, i+1)$, and when extension rule 1 applies (on a leaf edge), change the index pair on that leaf edge from (p, q) to $(p, q+1)$. It is easy to see inductively that q had to be i and hence the new label $(p, i+1)$ represents the correct new substring for that leaf edge.

By using an index pair to specify an edge-label, only two numbers are written on any edge, and since the number of edges is at most $2m - 1$, the suffix tree uses only $O(m)$ symbols and requires only $O(m)$ space. This makes it more plausible that the tree can actually be built in $O(m)$ time.[2] Although the fully implemented algorithm will not explicitly write a substring on an edge, we will still find it convenient to talk about "the substring or label on an edge or path" as if the explicit substring was written there.

[2] We make the standard RAM model assumption that a number with up to $\log m$ bits can be read, written, or compared in constant time.

6.1.5. Two more little tricks and we're done

We present two more implementation tricks that come from two observations about the way the extension rules interact in successive extensions and phases. These tricks, plus Lemma 6.1.2, will lead immediately to the desired linear time bound.

Observation 1: Rule 3 is a show stopper In any phase, if suffix extension rule 3 applies in extension j, it will also apply in all further extensions ($j+1$ to $i+1$) until the end of the phase. The reason is that when rule 3 applies, the path labeled $S[j..i]$ in the current tree must continue with character $S(i+1)$, and so the path labeled $S[j+1..i]$ does also, and rule 3 again applies in extensions $j+1, j+2, \ldots, i+1$.

When extension rule 3 applies, no work needs to be done since the suffix of interest is already in the tree. Moreover, a new suffix link needs to be added to the tree only after an extension in which extension rule 2 applies. These facts and Observation 1 lead to the following implementation trick.

Trick 2 End any phase $i+1$ the first time that extension rule 3 applies. If this happens in extension j, then there is no need to explicitly find the end of any string $S[k..i]$ for $k > j$.

The extensions in phase $i+1$ that are "done" after the first execution of rule 3 are said to be done ***implicitly.*** This is in contrast to any extension j where the end of $S[j..i]$ is explicitly found. An extension of that kind is called an ***explicit*** extension.

Trick 2 is clearly a good heuristic to reduce work, but it's not clear if it leads to a better worst-case time bound. For that we need one more observation and trick.

Observation 2: Once a leaf, always a leaf That is, if at some point in Ukkonen's algorithm a leaf is created and labeled j (for the suffix starting at position j of S), then that leaf will remain a leaf in all successive trees created during the algorithm. This is true because the algorithm has no mechanism for extending a leaf edge beyond its current leaf. In more detail, once there is a leaf labeled j, extension rule 1 will always apply to extension j in any successive phase. So once a leaf, always a leaf.

Now leaf 1 is created in phase 1, so in any phase i there is an initial sequence of consecutive extensions (starting with extension 1) where extension rule 1 or 2 applies. Let j_i denote the last extension in this sequence. Since any application of rule 2 creates a new leaf, it follows from Observation 2 that $j_i \leq j_{i+1}$. That is, the initial sequence of extensions where rule 1 or 2 applies cannot shrink in successive phases. This suggests an implementation trick that in phase $i+1$ avoids all explicit extensions 1 through j_i. Instead, only constant time will be required to do those extensions implicitly.

To describe the trick, recall that the label on any edge in an implicit suffix tree (or a suffix tree) can be represented by two indices p, q specifying the substring $S[p..q]$. Recall also that for any leaf edge of $\mathcal{I}_i$, index q is equal to i and in phase $i+1$ index q gets incremented to $i+1$, reflecting the addition of character $S(i+1)$ to the end of each suffix.

Trick 3 In phase $i+1$, when a leaf edge is first created and would normally be labeled with substring $S[p..i+1]$, instead of writing indices $(p, i+1)$ on the edge, write (p, e), where e is a symbol denoting "the current end". Symbol e is a ***global*** index that is set to $i+1$ once in each phase. In phase $i+1$, since the algorithm knows that rule 1 will apply in extensions 1 through j_i at least, it need do no additional explicit work to implement

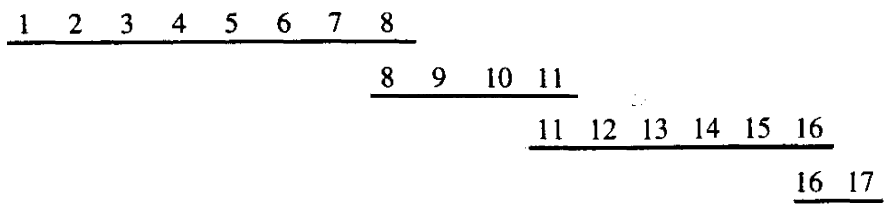

Figure 6.9: Cartoon of a possible execution of Ukkonen's algorithm. Each line represents a phase of the algorithm, and each number represents an explicit extension executed by the algorithm. In this cartoon there are four phases and seventeen explicit extensions. In any two consecutive phases, there is at most one index where the same explicit extension is executed in both phases.

those j_i extensions. Instead, it only does constant work to increment variable e, and then does explicit work for (some) extensions starting with extension $j_i + 1$.

The punch line

With Tricks 2 and **3,** explicit extensions in phase $i + 1$ (using algorithm SEA) are then only required from extension $j_i + 1$ until the first extension where rule 3 applies (or until extension $i + 1$ is done). All other extensions (before and after those explicit extensions) are done implicitly. Summarizing this, phase i + 1 is implemented as follows:

Single phase algorithm: SPA

Begin

1. Increment index e to $i + 1$. *(By* Trick **3** this correctly implements all implicit extensions 1 through j_i.)
2. Explicitly compute successive extensions (using algorithm SEA) starting at $j_i + 1$ until reaching the first extension j*where rule 3 applies or until all extensions are done in this phase. (By Trick 2, this correctly implements all the additional implicit extensions j*+ 1 through $i + 1$.)
3. **Set** j_{i+1} **to** $j^* - 1$, to prepare for the next phase.

End

Step *3* correctly sets j_{i+1} because the initial sequence of extensions where extension rule 1 or 2 applies must end at the point where rule **3** first applies.

The key feature of algorithm SPA is that phase i + 2 will *begin* computing explicit extensions with extension j^*, where j^* was the *last* explicit extension computed in phase $i + 1$. Therefore, two consecutive phases share *at most* one index (j^*) where an explicit extension is executed (see Figure 6.9). Moreover, phase i + 1 ends knowing where string $S[j^*..i + 1]$ ends, so the repeated extension of j^* in phase i + 2 can execute the suffix extension rule for j^* without any up-walking, suffix link traversals, or node skipping. That means the first explicit extension in any phase only takes constant time. It is now easy to prove the main result.

Theorem 6.1.2. *Using suffix links and implementation tricks* 1, 2, *and* 3, *Ukkanen's algorithm builds implicit suffix trees* $\mathcal{I}_1$ *through* $\mathcal{I}_m$ *in* $O(m)$ *total time.*

PROOF The time for all the implicit extensions in any phase is constant and so is $O(m)$ over the entire algorithm.

As the algorithm executes explicit extensions, consider an index $\bar{j}$ corresponding to the explicit extension the algorithm is currently executing. Over the entire execution of the algorithm, $\bar{j}$ never decreases, but it does remain the same between two successive phases.

Since there are only m phases, and $\bar{j}$ is bounded by m, the algorithm therefore executes only $2m$ explicit extensions. As established earlier, the time for an explicit extension is a constant plus some time proportional to the number of node skips it does during the down-walk in that extension.

To bound the total number of node skips done during all the down-walks, we consider (similar to the proof of Theorem 6.1.1) how the current node-depth changes during successive extensions, even extensions in different phases. The key is that the first explicit extension in any phase (after phase 1) begins with extension $\mathbf{j}^*$, which was the last explicit extension in the previous phase. Therefore, the current node-depth does not change between the end of one extension and the beginning of the next. But (as detailed in the proof of Theorem 6.1.1), in each explicit extension the current node-depth is first reduced by at most two (up-walking one edge and traversing one suffix link), and thereafter the down-walk in that extension increases the current node-depth by one at each node skip. Since the maximum node-depth is m, and there are only 2m explicit extensions, it follows (as in the proof of Theorem 6.1.1) that the maximum number of node skips done during all the down-walking (and not just in a single phase) is bounded by $O(m)$. All work has been accounted for, and the theorem is proved. □

6.1.6. Creating the true suffix tree

The final implicit suffix tree $\mathcal{I}_m$ can be converted to a true suffix tree in $O(m)$ time. First, add a string terminal symbol \$ to the end of S and let Ukkonen's algorithm continue with this character. The effect is that no suffix is now a prefix of any other suffix, so the execution of Ukkonen's algorithm results in an implicit suffix tree in which each suffix ends at a leaf and so is explicitly represented. The only other change needed is to replace each index e on every leaf edge with the number m. This is achieved by an $O(m)$-time traversal of the tree, visiting each leaf edge. When these modifications have been made, the resulting tree is a true suffix tree.

In summary,

Theorem 6.1.3. *Ukkonen's algorithm builds a true suffix tree for* S, *along with all its suffix links in* $O(m)$ *time.*

6.2. Weiner's linear-time suffix tree algorithm

Unlike Ukkonen's algorithm, Weiner's algorithm starts with the entire string S. However, like Ukkonen's algorithm, it enters one suffix at a time into a growing tree, although in a very different order. In particular, it first enters string $S(m)\$$ into the tree, then string $S[m-1..m]\$, \ldots,$ and finally, it enters the entire string $S\$$ into the tree.

Definition Suff_i denotes the suffix $S[i..m]$ of S starting in position i.

For example, Suff_1 is the entire string S, and Suff_m is the single character $S(m)$.

Definition Define $\mathcal{T}_i$ to be the tree that has $m - i + 2$ leaves numbered i through $m+1$ such that the path from the root to any leaf j $(i \le j \le m+1)$ has label $\text{Suff}_j\$$. That is, $\mathcal{T}_i$ is a tree encoding all and only the suffixes of string $S[i..m]\$$, so it is a suffix tree of string $S[i..m]\$$.

Weiner's algorithm constructs trees from $\mathcal{T}_{m+1}$ down to $\mathcal{T}_1$ (i.e., in decreasing order of i). We will first implement the method in a straightforward inefficient way. This will

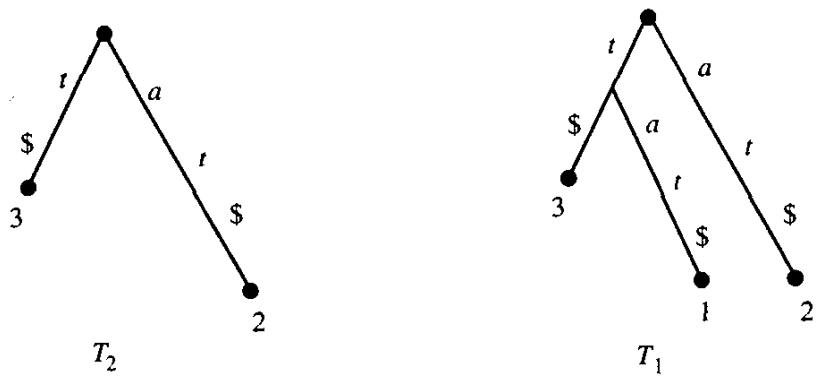

Figure 6.10: A step in the naive Weiner algorithm. The full string tat is added to the suffix tree for *at*. The edge labeled with the single character $ is omitted, since such an edge is part of every suffix tree.

serve to introduce and illustrate important definitions and facts. Then we will speed up the straightforward construction to obtain Weiner's linear-time algorithm.

6.2.1. A straightforward construction

The first tree $\mathcal{T}_{m+1}$ consists simply of a single edge out of the root labeled with the termination character \$. Thereafter, for each i from m down to 1, the algorithm constructs each tree $\mathcal{T}_i$ from tree $\mathcal{T}_{i+1}$ and character $S(i)$. The idea of the method is essentially the same as the idea for constructing keyword trees (Section 3.4), but for a different set of strings and without putting in backpointers. As the algorithm proceeds, each tree $\mathcal{T}_i$ will have the property that for any node v in $\mathcal{T}_i$, no two edges out of v have edge-labels beginning with the same character. Since $\mathcal{T}_{m+1}$ only has one edge out of the root, this is trivially true for $\mathcal{T}_{m+1}$. We assume inductively that this property is true for tree $\mathcal{T}_{i+1}$ and will verify that it holds for tree $\mathcal{T}_i$.

In general, to create $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, start at the root of $\mathcal{T}_{i+1}$ and walk as far as possible down a path whose label matches a prefix of $\text{Suff}_i\$$. Let R denote that path. In more detail, path R is found by starting at the root and explicitly matching successive characters of $\text{Suff}_i\$$ with successive characters along a unique path in $\mathcal{T}_{i+1}$. The matching path is unique, since at any node v of $\mathcal{T}_{i+1}$ no two edges out of v have edge-labels beginning with the same character. Thus the matching continues on at most one edge out of v. Ultimately, because no suffix is a prefix of another, no further match will be possible. If no node exists at that point, then create a new node there. In either case, refer to the node there (old or new) as w. Finally, add an edge out of w to a new leaf node labeled i, and label the new edge (w, i) with the remaining (unmatched) part of $\text{Suff}_i\$$. Since no further match had been possible, the first character on the label for edge (w, i) does not occur as the first character on any other edge out of w. Thus the claimed inductive property is maintained. Clearly, the path from the root to leaf i has label $\text{Suff}_i\$$. That is, that path exactly spells out string $\text{Suff}_i\$$, so tree $\mathcal{T}_i$ has been created.

For example, Figure 6.10 shows the transformation of $\mathcal{T}_2$ to $\mathcal{T}_1$ for the string ***tat.***

Definition For any position i, $Head(i)$ denotes the longest prefix of $S[i..m]$ that matches a substring of $S[i+1..m]\$$.

Note that $Head(i)$ could be the empty string. In fact, $Head(m)$ is always the empty string because $S[i+1..m]$ is the empty string when $i+1$ is greater than m and character $S(m) \neq \$$.

Since a copy of string *Head*(*i*) begins at some position between i + 1 and m, *Head*(*i*) is also a prefix of Suff_k for some k > i. It follows that *Head*(*i*) is the longest prefix (possibly empty) of Suff_i that is a label on some path from the root in tree $\mathcal{T}_{i+1}$.

The above straightforward algorithm to build $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ can be described as follows:

Naive Weiner algorithm

1. Find the end of the path labeled *Head*(*i*) in tree $\mathcal{T}_{i+1}$.
2. If there is no node at the end of *Head*(*i*) then create one, and let w denote the node (created or not) at the end of *Head*(*i*). If w is created at this point, splitting an existing edge, then split its existing edge-label so that w has node-label *Head*(*i*). Then, create a new leaf numbered i and a new edge (w, i) labeled with the remaining characters of $\text{Suff}_i\$$. That is, the new edge-label should be the last $m - i + 1 - |Head(i)|$ characters of Suff_i, followed by the termination symbol \$.

6.2.2. Toward a more efficient implementation

It should be clear that the final suffix tree $\mathcal{T} = \mathcal{T}_1$ is constructed in $O(m^2)$ time by this straightforward approach. Clearly, the difficult part of the algorithm is finding *Head*(*i*), since step 2 takes only constant time for any i. So, to speed up the algorithm, we will need a more efficient way to find *Head*(*i*). But, as in the discussion of Ukkonen's algorithm, a linear time bound is not possible if edge-labels are explicitly written on the tree. Instead, each edge-label is represented by two indices indicating the start and end positions of the labeling substring. The reader should review Section 6.1.4 at this point.

It is easy to implement Weiner's algorithm using an index pair to label an edge. When inserting Suff_i, suppose the algorithm has matched up to the kth character on an edge (u, z) labeled with interval [s, t], but the next character is a mismatch. A new node w is created dividing (u, z) into two edges, (u, w) and (w, z), and a new edge is also created from w to leaf i. Edge (u, w) gets labeled [s, s + k − 1], edge (w, z) gets labeled [s + k, t], and edge (w, i) gets labeled [i + d(w), m]\$, where d(w) is the string-depth (number of characters) of the path from the root down to node w. These string-depths can easily be created and maintained as the tree is being built, since d(w) = d(u) + k. The string-depth of a leaf i is m − i + 1.

Finding *Head*(*i*) efficiently

We now return to the central issue of how to find *Head*(*i*) efficiently. The key to Weiner's algorithm are two vectors kept at each nonleaf node (including the root). The first vector is called the indicator vector I and the second is called the *link* vector L. Each vector is of length equal to the size of the alphabet, and each is indexed by the characters of the alphabet. For example, for the English alphabet augmented with \$, each link and indicator vector will be of length 27.

The link vector is essentially the reverse of the suffix link in Ukkonen's algorithm, and the two links are used in similar ways to accelerate traversals inside the tree.

The indicator vector is a bit vector so its entries are just 0 or 1, whereas each entry in the link vector is either null or is a pointer to a tree node. Let $I_v(x)$ specify the entry of the indicator vector at node v indexed by character x. Similarly, let $L_v(x)$ specify the entry of the link vector at node v indexed by character x.

The vectors I and L have two crucial properties that will be maintained inductively throughout the algorithm:

- For any (single) character x and any node u, $I_u(x) = 1$ in $\mathcal{T}_{i+1}$ if and only if there is a *path* from the root of $\mathcal{T}_{i+1}$ labeled $x\alpha$, where α is the path-label of node u. The path labeled $x\alpha$ need not end at a node.
- For any character x, $L_u(x)$ in $\mathcal{T}_{i+1}$ points to (internal) **node** $\overline{u}$ in $\mathcal{T}_{i+1}$ if and only if $\overline{u}$ has path-label xa, where u has path-label a. Otherwise $L_u(x)$ is null.

For example, in the tree in Figure 5.1 (page 91) consider the two internal nodes u and w with path-labels a and xa respectively. Then $I_u(x) = 1$ for the specific character x, and $L_u(x) = w$. Also, $I_w(b) = 1$, but $L_w(b)$ is null.

Clearly, for any node u and any character x, $L_u(x)$ is nonnull only if $I_u(x) = 1$, but the converse is not true. It is also immediate that if $I_u(x) = 1$ then $I_v(x) = 1$ for every ancestor node v of u.

Tree $\mathcal{T}_m$ has only one nonleaf node, namely the root r. In this tree we set $I_r(S(m))$ to one, set $I_r(x)$ to zero for every other character x, and set all the link entries for the root to null. Hence the above properties hold for $\mathcal{T}_m$. The algorithm will maintain the vectors as the tree changes, and we will prove inductively that the above properties hold for each tree.

6.2.3. The basic idea of Weiner's algorithm

Weiner's algorithm uses the indicator and link vectors to find $Head(i)$ and to construct $\mathcal{T}_i$ more efficiently. The algorithm must take care of two degenerate cases, but these are not much different than the general "good" case where no degeneracy occurs. We first discuss how to construct $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ in the good case, and then we handle the degenerate cases.

The algorithm in the good case

We assume that tree $\mathcal{T}_{i+1}$ has just been constructed and we now want to build $\mathcal{T}_i$. The algorithm starts at leaf $i+1$ of $\mathcal{T}_{i+1}$ (the leaf for Suff_{i+1}) and walks toward the root looking for the first node v, if it exists, such that $I_v(S(i)) = 1$. If found, it then continues from v walking upwards toward the root searching for the first node v' it encounters (possibly v) where $L_{v'}(S(i))$ is nonnull. By definition, $L_{v'}(S(i))$ is nonnull only if $I_{v'}(S(i)) = 1$, so if found, v' will also be the first node encountered on the walk from leaf $i+1$ such that $L_{v'}(S(i))$ is nonnull. In general, it may be that neither v nor v' exist or that v exists but v' does not. Note, however, that v or v' may be the root.

The "good case" is that both v and v' do exist.

Let l_i be the number of characters on the path between v' and v, and if $l_i > 0$ then let c denote the first of these l_i characters.

Assuming the good case, that both v and v' exist, we will prove below that if node v has path-label α then $Head(i)$ is precisely string $S(i)\alpha$. Further, we will prove that when $L_{v'}(S(i))$ points to node v'' in $\mathcal{T}_{i+1}$, $Head(i)$ either ends at v'', if $l_i = 0$, or else it ends *exactly* l_i characters below v'' on an edge out of v''. So in either case, $Head(i)$ can be found in constant time after v' is found.

Theorem 6.2.1. Assume that node v has been found by the *algorithm and* that it has path-label a. Then the string $Head(i)$ is exactly $S(i)\alpha$.

PROOF $Head(i)$ is the longest prefix of Suff_i that is also a prefix of Suff_k for some $k > i$. Since v was found with $I_v(S(i)) = 1$ there is a path in $\mathcal{T}_{i+1}$ that begins with $S(i)$, so $Head(i)$ is at least one character long. Therefore, we can express $Head(i)$ as $S(i)\beta$, for some (possibly empty) string β.

Suff_i and Suff_k both begin with string $Head(i) = S(i)\beta$ and differ after that. For concreteness, say Suff_i begins $S(i)\beta a$ and Suff_k begins $S(i)\beta b$. But then Suff_{i+1} begins βa and Suff_{k+1} begins βb. Both i + 1 and k + 1 are greater than or equal to i + 1 and less than or equal to m, so both suffixes are represented in tree $\mathcal{T}_{i+1}$. Therefore, in tree $\mathcal{T}_{i+1}$, there must be a path from the root labeled β (possibly the empty string) that extends in two ways, one continuing with character a and the other with character b. Hence there is a node u in $\mathcal{T}_{i+1}$ with path-label β, and $l_u(S(i)) = 1$ since there is a path (namely, an initial part of the path to leaf k) labeled $S(i)\beta$ in $\mathcal{T}_{i+1}$. Further, node u must be on the path to leaf i + 1 since β is a prefix of Suff_{i+1}.

Now $l_v(S(i)) = 1$ and v has path-label a, so $Head(i)$ must begin with $S(i)\alpha$. That means that α is a prefix of β and so node u, with path label β, must either be v or below v on the path to leaf i + 1. However, if u # v then u would be a node below v on the path to leaf i + 1, and $l_u(S(i)) = 1$. This contradicts the choice of node v, so $v = u, \alpha = \beta$, and the theorem is proved. That is, $Head(i)$ is exactly the string $S(i)\alpha$. □

Note that in Theorem 6.2.1 and its proof we only assume that node v exists. No assumption about v' was made. This will be useful in one of the degenerate cases examined later.

Theorem 6.2.2. Assume both v and v' have been found and $L_{v'}(S(i))$ points to node v''. *If* $l_i = 0$ then $Head(i)$ ends at v''; *otherwise* it ends after exactly l_i characters on a single edge ***out of*** v".

PROOF Since v' is on the path to leaf i + 1 and $L_{v'}(S(i))$ points to node v'', the path from the root labeled $Head(i)$ must include v''. By Theorem 6.2.1 $Head(i) = S(i)\alpha$, so $Head(i)$ must end exactly l_i characters below v''. Thus, when $l_i = 0$, $Head(i)$ ends at v''. But when $l_i > 0$, there must be an edge e = (v'', z) out of v'' whose label begins with character c (the first of the l_i characters on the path from v' to v) in $\mathcal{T}_{i+1}$.

Can $Head(i)$ extend down to node z (i.e., to a node below v'')? Node z must be a branching node, for if it were a leaf then some suffix Suff_k, for k > i, would be a prefix of Suff_i, which is not possible. Let z have path-label $S(i)\gamma$. If $Head(i)$ extends down to branching node z, then there must be two substrings starting at or after position i + 1 of S that both begin with string y. Therefore, there would be a node z' with path-label y in $\mathcal{T}_{i+1}$. Node z' would then be below v' on the path to leaf i + 1, contradicting the selection of v'. So $Head(i)$ must not reach z and must end in the interior of edge e. In particular, it ends exactly l_i characters from v'' on edge e. □

Thus when $l_i = 0$, we know $Head(i)$ ends at v'', and when $l_i > 0$, we find $Head(i)$ from v" by examining the edges out of v'' to identify that unique edge e whose first character is c. Then $Head(i)$ ends exactly l_i characters down e from v''. Tree $\mathcal{T}_i$ is then constructed by subdividing edge e, creating a node w at this point, and adding a new edge from w to leaf i labeled with the remainder of Suff_i. The search for the correct edge out of v'' takes only constant time since the alphabet is fixed.

In summary, when v and v' exist, the above method correctly creates $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$, although we must still discuss how to update the vectors. Also, it may not yet be clear at this point why this method is more efficient than the naive algorithm for finding $Head(i)$. That will come later. Let us first examine how the algorithm handles the degenerate cases when v and v' do not both exist.

The two degenerate cases

The two degenerate cases are that node v (and hence node v') does not exist or that v exists but v' does not. We will see how to find $Head(i)$ efficiently in these two cases. Recall that r denotes the root node.

Case 1 $I_r(S(i)) = 0$.

In this case the walk ends at the root and no node v was found. It follows that character $S(i)$ does not appear in any position greater than i, for if it did appear, then some suffix in that range would begin with $S(i)$, some path from the root would begin with $S(i)$, and $I_r(S(i))$ would have been 1. So when $I_r(S(i)) = 0$, $Head(i)$ is the empty string and ends at the root.

Case 2 $I_v(S(i)) = 1$ for some v (possibly the root), but v' does not exist.

In this case the walk ends at the root with $L_r(S(i))$ null. Let t_i be the number of characters from the root to v. From Theorem 6.2.1 $Head(i)$ ends exactly $t_i + 1$ characters from the root. Since v exists, there is some edge e = (r, z) whose edge-label begins with character $S(i)$. This is true whether $t_i = 0$ or $t_i > 0$.

If $t_i = 0$ then $Head(i)$ ends after the first character, $S(i)$, on edge e.

Similarly, if $t_i > 0$ then $Head(i)$ ends exactly $t_i + 1$ characters from the root on edge e. For suppose $Head(i)$ extends all the way to some child z (or beyond). Then exactly as in the proof of Theorem 6.2.2, z must be a branching node and there must be a node z' below the root on the path to leaf i + 1 such that $L_{z'}(S(i))$ is nonnull, which would be a contradiction. So when $t_i > 0$, $Head(i)$ ends exactly $t_i + 1$ characters from the root on the edge e out of the root. This edge can be found from the root in constant time since its first character is S(i).

In either of these degenerate cases (as in the good case), $Head(i)$ is found in constant time after the walk reaches the root. After the end of $Head(i)$ is found and w is created or found, the algorithm proceeds exactly as in the good case.

Note that degenerate Case 2 is very similar to the "good case when both v and v' were found, but differs in a small detail because $Head(i)$ is found $t_i + 1$ characters down on e rather than t_i characters down (the natural analogue of the good case).

6.2.4. The full algorithm for creating $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$

Incorporating all the cases gives the following algorithm:

Weiner's Tree extension

1. Start at leaf i + 1 of $\mathcal{T}_{i+1}$ (the leaf for $Suff_{i+1}$) and walk toward the root searching for the first node v on the walk such that $I_v(S(i)) = 1$.

2. If the root is reached and $I_r(S(i)) = 0$, then $Head(i)$ ends at the root. Go to Step 4.

3. Let v be the node found (possibly the root) such that $I_v(S(i)) = 1$. Then continue walking upward searching for the first node v' (possibly v itself) such that $L_{v'}(S(i))$ is nonnull.

3a. If the root is reached and $L_r(S(i))$ is null, let t_i be the number of characters on the path between the root and v. Search for the edge e out of the root whose edge-label begins with $S(i)$. $Head(i)$ ends exactly $t_i + 1$ characters from the root on edge e.

Else {when the condition in 3a does not hold)

3b. If v' was found such that $L_{v'}(S(i))$ is nonnull, say v", then follow the link (for $S(i)$) to v". Let l_i be the number of characters on the path from v' to v and let c be the first character

on this path. If $l_i = 0$ then $Head(i)$ ends at v''. Otherwise, search for the edge e out of v'' whose first character is c. $Head(i)$ ends exactly l_i characters below v" on edge e.

4. If a node already exists at the end of $Head(i)$, then let w denote that node; otherwise, create a node w at the end of $Head(i)$. Create a new leaf numbered i; create a new edge (w, i) labeled with the remaining substring of $Suff_i$ (i.e., the last m − i + 1 − $|Head(i)|$ characters of Suff,), followed with the termination character \$. Tree T_i has now been created.

Correctness

It should be clear from the proof of Theorems 6.2.1 and 6.2.2 and the discussion of the degenerate cases that the algorithm correctly creates tree T_i from T_{i+1}, although before it can create T_{i-1}, it must update the I and L vectors,

How to update the vectors

After finding (or creating) node w, we must update the I and L vectors so that they are correct for tree T_i. If the algorithm found a node v such that $I_v(S(i)) = 1$, then by Theorem 6.2.1 node w has path-label $S(i)\alpha$ in T_i, where node v has path-label α. In this case, $L_v(S(i))$ should be set to point to w in T_i. This is the only update needed for the link vectors since only one node can point via a link vector to any other node and only one new node was created. Furthermore, if node w is newly created, all its link entries for T_i should be null. To see this, suppose to the contrary that there is a node u in T_i with path-label $xHead(i)$, where w has path-label $Head(i)$. Node u cannot be a leaf because $Head(i)$ does not contain the character \$. But then there must have been a node in T_{i+1} with path-label $Head(i)$, contradicting the fact that node w was inserted into T_{i+1} to create T_i. Consequently, there is no node in T_i with path-label $xHead(i)$ for any character x and all the L vector values for w should be null.

Now consider the updates needed to the indicator vectors for tree T_i. For every node u on the path from the root to leaf i + 1, $I_u(S(i))$ must be set to 1 in T_i since there is now a path for string $Suff_i$ in T_i. It is easy to establish inductively that if a node v with $I_v(S(i)) = 1$ is found during the walk from leaf $i + 1$, then every node u above v on the path to the root already has $I_u(S(i)) = 1$. Therefore, only the indicator vectors for the nodes below v on the path to leaf i + 1 need to be set. If no node v was found, then all nodes on the path from i + 1 to the root were traversed and all of these nodes must have their indicator vectors updated. The needed updates for the nodes below v can be made during the search for v (i.e., no separate pass is needed). During the walk from leaf $i + 1$, $I_u((S(i))$ is set to 1 for every node u encountered on the walk. The time to set these indicator vectors is proportional to the time for the walk.

The only remaining update is to set the I vector for a newly created node w created in the interior of an edge $e = (v'', z)$.

Theorem 6.2.3. When *a new* node w is created in the interior of an edge (v'', z) the indicator vectorfor w *should* be copiedfrom the indicator vectorfor z.

PROOF It is immediate that if $I_z(x) = 1$ then $I_w(x)$ must also be 1 in T_i. But can it happen that $I_w(x)$ should be 1 and yet $I_z(x)$ is set to 0 at the moment that w is created? We will see that it cannot.

Let node z have path-label γ, and of course node w has path-label $Head(i)$, a prefix of γ. The fact that there are no nodes between u and z in T_{i+1} means that every suffix from $Suff_m$ down to $Suff_{i+1}$ that begins with string $Head(i)$ must actually begin with the longer

string γ. Hence in $\mathcal{T}_{i+1}$ there can be a path labeled $xHead(i)$ only if there is also a path labeled x y, and this holds for any character x. Therefore, if there is a path in $\mathcal{T}_i$ labeled $xHead(i)$ (the requirement for $I_w(x)$ to be 1) but no path x y, then the hypothesized string $xHead(i)$ must begin at character i of S. That means that Suff_{i+1} must begin with the string $Head(i)$. But since w has path-label $Head(i)$, leaf $i+1$ must be below w in $\mathcal{T}_i$ and so must be below z in $\mathcal{T}_{i+1}$. That is, z is on the root to $i+1$ path. However, the algorithm to construct $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ starts at leaf i + 1 and walks toward the root, and when it finds node v or reaches the root, the indicator entry for x has been set to 1 at every node on the path from the leaf $i+1$. The walk finishes before node w is created, and so it cannot be that $I_z(x) = 0$ at the time when w is created. So if path $xHead(i)$ exists in $\mathcal{T}_i$, then $I_z(x) = 1$ at the moment w is created, and the theorem is proved. □

6.2.5. Time analysis of Weiner's algorithm

The time to construct $\mathcal{T}_i$ from $\mathcal{T}_{i+1}$ and update the vectors is proportional to the time needed during the walk from leaf i + 1 (ending either at v' or the root). This walk moves from one node to its parent, and assuming the usual parent pointers, only constant time is used to move between nodes. Only constant time is used to follow a L link pointer, and only constant time is used after that to add w and edge (w, i). Hence the time to construct $\mathcal{T}_i$ is proportional to the number of nodes encountered on the walk from leaf i + 1.

Recall that the node-depth of a node v is the number of nodes on the path in the tree from the root to v.

For the time analysis we imagine that as the algorithm runs we keep track of what node has most recently been encountered and what its node-depth is. Call the node-depth of the most recently encountered node the current node-depth. For example, when the algorithm begins, the current node-depth is one and just after $\mathcal{T}_m$ is created the current node-depth is two. Clearly, when the algorithm walks up a path from a leaf the current node-depth decreases by one at each step. Also, when the algorithm is at node v" (or at the root) and then creates a new node w below v" (or below the root), the current node-depth increases by one. The only question remaining is how the current node-depth changes when a link pointer is traversed from a node v' to v".

Lemma 6.2.1. When the algorithm traverses a link pointer from a node v' to a node v" in $\mathcal{T}_{i+1}$, the current node-depth increases by at *most* one.

PROOF Let u be a nonroot node in $\mathcal{T}_{i+1}$ on the path from the root to v", and suppose u has path-label $S(i)\alpha$ for some nonempty string a. All nodes on the root-to-v" path are of this type, except for the single node (if it exists) with path-label $S(i)$. Now $S(i)\alpha$ is the prefix of Suff_i and of Suff_k for some k > i, and this string extends differently in the two cases. Since v' is on the path from the root to leaf $i+1$, a is a prefix of Suff_{i+1}, and there must be a node (possibly the root) with path-label α on the path to v' in $\mathcal{T}_{i+1}$. Hence the path to v' has a node corresponding to every node on the path to v", except the node (if it exists) with path-label $S(i)$. Hence the depth of v'' is at most one more than the depth of v', although it could be less. □

We can now finish the time analysis.

Theorem 6.2.4. Assuming *a finite* alphabet, Weiner's algorithm *constructs* the *suffix* tree for a string of length m *in* $O(m)$ time.

PROOF The current node-depth can increase by one each time a new node is created and each time a link pointer is traversed. Hence the total number of increases in the current node-depth is at most 2m. It follows that the current node-depth can also only decrease at most 2m times since the current node-depth starts at zero and is never negative. The current node-depth decreases at each move up the walk, so the total number of nodes visited during all the upward walks is at most $2m$. The time for the algorithm is proportional to the total number of nodes visited during upward walks, so the theorem is proved. □

6.2.6. Last comments about Weiner's algorithm

Our discussion of Weiner's algorithm establishes an important fact about suffix trees, regardless of how they are constructed:

Theorem 6.2.5. *If* v *is a node in the suffix tree labeled by the string* xa*, where* x *is a single character; then there is a node in the tree labeled with the string* α.

This fact was also established as Corollary 6.1.2 during the discussion of Ukkonen's algorithm.

6.3. McCreight's suffix tree algorithm

Several years after Weiner published his linear-time algorithm to construct a suffix tree for a string S, McCreight [318] gave a different method that also runs in linear time but is more space efficient in practice. The inefficiency in Weiner's algorithm is the space it needs for the indicator and link vectors, I and L, kept at each node. For a fixed alphabet, this space is considered linear in the length of S, but the space used may be large in practice. McCreight's algorithm does not need those vectors and hence uses less space.

Ukkonen's algorithm also does not use the vectors I and L of Weiner's algorithm, and it has the same space efficiency as McCreight's algorithm.[3] In fact, the fully implemented version of Ukkonen's algorithm can be seen as a somewhat disguised version of McCreight's algorithm. However, the high-level organization of Ukkonen and McCreight's algorithms are quite different, and the connection between the algorithms is not obvious. That connection was suggested by Ukkonen [438] and made explicit by Giegerich and Kurtz [178]. Since Ukkonen's algorithm has all the advantages of McCreight's, and is simpler to describe, we will only introduce McCreight's algorithm at the high level.

McCreight's algorithm at the high level

McCreight's algorithm builds the suffix tree T for rn-length string S by inserting the suffixes in order, one at a time, starting from suffix one (i.e., the complete string S). (This is opposite to the order used in Weiner's algorithm, and it is superficially different from Ukkonen's algorithm.) It builds a tree encoding all the suffixes of S starting at positions 1 through i + 1, from the tree encoding all the suffixes of S starting at positions 1 through i.

The naive construction method is immediate and runs in $O(m^2)$ time. Using suffix links and the skip/count trick, that time can be reduced to $O(m)$. We leave this to the interested reader to work out.

[3] **The space requirements For Ukkonen and McCreight's algorithms are determined by the need to represent and move around the tree quickly. We will be much more precise about space and practical impfementation issues in Section 6.5.**

6.4. Generalized suffix tree for a set of strings

We have so far seen methods to build a suffix tree for a single string in linear time. Those methods are easily extended to represent the suffixes of a set $\{S_1, S_2, \ldots, S_n\}$ of strings. Those suffixes are represented in a tree called a generalized suffix tree, which will be used in many applications.

A conceptually easy way to build a generalized suffix tree is to append a different end of string marker to each string in the set, then concatenate all the strings together, and build a suffix tree for the concatenated string. The end of string markers must be symbols that are not used in any of the strings. The resulting suffix tree will have one leaf for each suffix of the concatenated string and is built in time proportional to the sum of all the string lengths. The leaf numbers can easily be converted to two numbers, one identifying a string S_i and the other a starting position in S_i.

One defect with this way of constructing a generalized suffix tree is that the tree represents substrings (of the concatenated string) that span more than one of the original strings. These "synthetic" suffixes are not generally of interest. However, because each end of string marker is distinct and is not in any of the original strings, the label on any path from the root to an internal node must be a substring of one of the original strings. Hence by reducing the second index of the label on leaf edges, without changing any other parts of the tree, all the unwanted synthetic suffixes are removed.

Under closer examination, the above method can be simulated without first concatenating the strings. We describe the simulation using Ukkonen's algorithm and two strings S_1 and S_2, assumed to be distinct. First build a suffix tree for S_1 (assuming an added terminal character). Then starting at the root of this tree, match S_2 (again assuming the same terminal character has been added) against a path in the tree until a mismatch occurs. Suppose that the first i characters of S_2 match. The tree at this point encodes all the suffixes of S_1, and it implicitly encodes every suffix of the string $S_2[1..i]$. Essentially, the first i phases of Ukkonen's algorithm for S_2 have been executed on top of the tree for S_1. So, with that current tree, resume Ukkonen's algorithm on S_2 in phase i + 1. That is, walk up at most one node from the end of $S_2[1..i]$, etc, When S_2 is fully processed the tree will encode all the suffixes of S_1 and all the suffixes of S_2 but will have no synthetic suffixes. Repeating this approach for each of the strings in the set creates the generalized suffix tree in time proportional to the sum of the lengths of all the strings in the set.

There are two minor subtleties with the second approach. One is that the compressed labels on different edges may refer to different strings. Hence the number of symbols per edge increases from two to three, but otherwise causes no problem. The second subtlety is that suffixes from two strings may be identical, although it will still be true that no suffix is a prefix of any other. In this case, a leaf must indicate all of the strings and starting positions of the associated suffix.

As an example, if we add the string ***bnbxba*** to the tree for ***xabxa*** (shown in Figure 6.1), the result is the generalized suffix tree shown in Figure 6.11.

6.5. Practical implementation issues

The implementation details already discussed in this chapter turn naive, quadratic (or even cubic) time algorithms into algorithms that run in $O(m)$ worst-case time, assuming a fixed alphabet Σ. But to make suffix trees truly practical, more attention to implementation is needed, particularly as the size of the alphabet grows. There are problems nicely solved

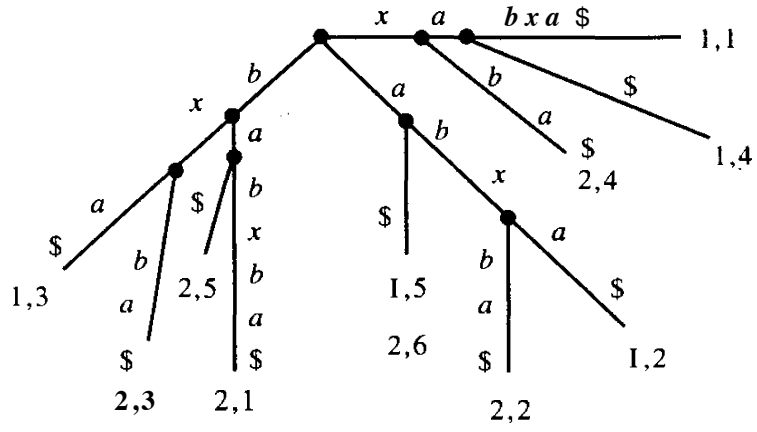

Figure 6.11: Generalized suffix tree for strings $S_1 = xabxa$ and $S_2 = babxba$. The first number at a leaf indicates the string; the second number indicates the starting position of the suffix in that string.

in theory by suffix trees, where the typical string size is in the hundreds of thousands, or even millions, and/or where the alphabet size is in the hundreds. For those problems, a "linear" time and space bound is not sufficient assurance of practicality. For large trees, paging can also be a serious problem because the trees do not have nice locality properties. Indeed, by design, suffix links allow an algorithm to move quickly from one part of the tree to a distant part of the tree. This is great for worst-case time bounds, but it is horrible for paging if the tree isn't entirely in memory. Consequently, implementing suffix trees to reduce practical space use can be a serious concern.[4] The comments made here for suffix trees apply as well to keyword trees used in the Aho–Corasick method.

The main design issues in all three algorithms are how to represent and search the branches out of the nodes of the tree and how to represent the indicator and link vectors in Weiner's algorithm. A practical design must balance the constraints of space against the need for speed, both in building the tree and in using it afterwards. We will discuss representing tree edges, since the vector issues for Weiner's algorithm are identical.

There are four basic choices possible to represent branches. The simplest is to use an *array* of size $\Theta(|\Sigma|)$ at each nonleaf node v. The m a y at v is indexed by single characters of the alphabet; the cell indexed by character x has a pointer to a child of v if there is an edge out of v whose edge-label begins with character x and is otherwise null. If there is such an edge, then the cell should also hold the two indices representing its edge-label. This array allows constant-time random accesses and updates and. although simple to program, it can use an impractical amount of space as $|\Sigma|$ and m get large.

An alternative to the array is to use a ***linked list*** at node v of characters that appear at the beginning of edge-labels out of ***v.*** When a new edge from v is added to the tree, a new character (the first character on the new edge label) is added to the list. Traversals from node v are implemented by sequentially searching the list for the appropriate character. Since the list is searched sequentially it costs no more to keep it in ***sorted order.*** This somewhat reduces the average time to search for a given character and thus speeds up (in practice) the construction of the tree. The key point is that it allows a faster termination of a search for a character that is not in the list. Keeping the list in sorted order will be particularly useful in some of applications of suffix trees to be discussed later.

[4] **A very different approach to limiting space. based on changing the suffix tree into a different data structure called a *suffix array.* will be discussed in Section 7.14.**

Keeping a linked list at node v works well if the number of children of v is small, but in worst-case adds time $|\Sigma|$ to every node operation. The $O(m)$ worst-case time bounds are preserved since $|\Sigma|$ is assumed to be fixed, but if the number of children of v is large then little space is saved over the array while noticeably degrading performance.

A third choice, a compromise between space and speed, is to implement the list at node v as some sort of ***balanced tree*** [10]. Additions and searches then take $O(\log k)$ time and $O(k)$ space, where k is the number of children of v. Due to the space and programming overhead of these methods, this alternative makes sense only when k is fairly large.

The final choice is some sort of ***hashing scheme.*** Again, the challenge is to find a scheme balancing space with speed, but for large trees and alphabets hashing is very attractive at least for some of the nodes. And, using perfect hashing techniques [167] the linear worst-case time bound can even be preserved.

When m and Σ are large enough to make implementation difficult, the best design is probably a mixture of the above choices. Nodes near the root of the tree tend to have the most children (the root has a child for every distinct character appearing in S), and so arrays are a sensible choice at those nodes. In addition, if the tree is dense for several levels below the root, then those levels can be condensed and eliminated from the explicit tree. For example, there are 20^5 possible amino acid substrings of length five. Every one of these substrings exists in some known protein sequence already in the databases. Therefore, when implementing a suffix tree for the protein database, one can replace the first five levels of the tree with a five-dimensional array (indexed by substrings of length five), where an entry of the array points to the place in the remaining tree that extends the five-tuple, The same idea has been applied [320] to depth seven for DNA data. Nodes in the suffix tree toward the leaves tend to have few children and lists there are attractive. At the extreme, if w is a leaf and v is its parent, then information about w may be brought up to v, removing the need for explicit representation of the edge (v, w) or the node w. Depending on the other implementation choices, this can lead to a large savings in space since roughly half the nodes in a suffix tree are leaves. A suffix tree whose leaves are deleted in this way is called a ***position tree.*** In a position tree, there is a one-to-one correspondence between leaves of the tree and substrings that are uniquely occurring in S.

For nodes in the middle of a suffix tree, hashing or balanced trees may be the best choice. Fortunately, most large suffix trees are used in applications where S is fixed (a dictionary or database) for some time and the suffix tree will be used repeatedly. In those applications, one has the time and motivation to experiment with different implementation choices. For a more in-depth look at suffix tree implementation issues, and other suggested variants of suffix trees, see [23].

Whatever implementation is selected, it is clear that a suffix tree for a string will take considerably more space than the representation of the string itself.[5] Later in the book we will discuss several problems involving two (or more) strings P and T, where two $O(|P| + |T|)$ time solutions exist, one using a suffix tree for P and one using a suffix tree for T. We will also have examples where equally time-efficient solutions exist, but where one uses a generalized suffix tree for two or more strings and the other uses just a suffix tree for the smaller string. In asymptotic worst-case time and space, neither approach is superior to the other, and usually the approach that builds the larger tree is conceptually simpler. However, when space is a serious practical concern (and in many problems, including

[5] **Although, we have built suffix trees for DNA and amino acid strings more than one million characters long that can be completely contained in the main memory of a moderate-size workstation.**

many in molecular biology, space is more of a constraint than is time), the size of the suffix tree for a string may dictate using the solution that builds the smaller suffix tree. So despite the added conceptual burden, we will discuss such space-reducing alternatives in some detail throughout the book.

6.5.1. Alphabet independence: all linears are equal, but some are more equal than others

The key implementation problems discussed above are all related to multiple edges (or links) at nodes. These are influenced by the size of the alphabet Σ – the larger the alphabet, the larger the problem. For that reason, some people prefer to explicitly reflect the alphabet size in the time and space bounds of keyword and suffix tree algorithms. Those people usually refer to the construction time for keyword or suffix trees as O(m log $|\Sigma|$), where m is the size of all the patterns in a keyword tree or the size of the string in a suffix tree. More completely, the Aho–Corasick, Weiner, Ukkonen, and McCreight algorithms all either require $\Theta(m|\Sigma|)$ space, or the $O(m)$ time bound should be replaced with the minimum of O(m log m) and $O(m \log |\Sigma|)$. Similarly, searching for a pattern P using a suffix tree can be done with $O(|P|)$ comparisons only if we use $\Theta(m|\Sigma|)$ space; otherwise we must allow the minimum of $O(|P| \log m)$ and $O(|P| \log |\Sigma|)$ comparisons during a search for P.

In contrast, the exact matching method using Z values has worst-case space and comparison requirements that are ***alphabet*** independent – the worst-case number of comparisons (either characters or numbers) used to compute Z values is uninfluenced by the size of the alphabet. Moreover, when two characters are compared, the method only checks whether the characters are equal or unequal, not whether one character precedes the other in some ordering. Hence no prior knowledge about the alphabet need be assumed. These properties are also true of the Knuth-Morris-Pratt and the Boyer–Moore algorithms. The alphabet independence of these algorithms makes their linear time and space bounds superior, in some people's view, to the linear time and space bounds of keyword and suffix tree algorithms: "All linears are equal but some are more equal than others". Alphabet-independent algorithms have also been developed for a number of problems other than exact matching. Two-dimensional exact matching is one such example. The method presented in Section 3.5.3 for two-dimensional matching is based on keyword trees and hence is not alphabet independent. Nevertheless, alphabet-independent solutions for that problem have been developed. Generally, alphabet-independent methods are more complex than their coarser counterparts. In this book we will not consider alphabet-independence much further, although we will discuss other approaches to reducing space that can be employed if large alphabets cause excessive space use.

6.6. Exercises

1. **Construct an infinite family of strings over a fixed alphabet, where the total length of the edge-labels on their suffix trees grows faster than $\Theta(m)$ (m is the length of the string). That is, show that linear-time suffix tree algorithms would be impossible if edge-labels were written explicitly on the edges.**
2. **In the text, we first introduced Ukkonen's algorithm at a high level and noted that it could be implemented in $O(m^3)$ time. That time was then reduced to $O(m^2)$ with the use of suffix links and the skip/count trick. An alternative way to reduce the $O(m^3)$ time to $O(m^2)$ (without suffix links or skip/count) is to keep a pointer to the end of each suffix of $S[1..i]$.**

Then Ukkonen's high-level algorithm could visit all these ends and create $\mathcal{I}_{i+1}$ from $\mathcal{I}_i$ in $O(i)$ time, so that the entire algorithm would run in $O(m^2)$ time. Explain this in detail.

3. The relationship between the suffix tree for a string S and for the reverse string S^r is not obvious. However, there is a significant relationship between the two trees. Find it, state it, and prove it.

   Hint: Suffix links help.

4. Can Ukkonen's algorithm be implemented in linear time without using suffix links? The idea is to maintain, for each index i, a pointer to the node in the current implicit suffix tree that is closest to the end of suffix i.

5. In Trick 3 of Ukkonen's algorithm, the symbol "e" is used as the second index on the label of every leaf edge, and in phase $i + 1$ the global variable e is set to $i + 1$. An alternative to using "e" is to set the second index on any leaf edge to m (the total length of S) at the point that the leaf edge is created. In that way, no work is required to update that second index. Explain in detail why this is correct, and discuss any disadvantages there may be in this approach, compared to using the symbol "e".

6. Ukkonen's algorithm builds all the implicit suffix trees I_1 through I_m in order and ***on-line,*** all in $O(m)$ time. Thus it can be called a linear-time on-line algorithm to construct implicit suffix trees.

   (Open question) Find an on-line algorithm running in $O(m)$ total time that creates all the ***true*** suffix trees. Since the time taken to explicitly store these trees is $0(m^2)$, such an algorithm would (like Ukkonen's algorithm) update each tree without saving it.

7. Ukkonen's algorithm builds all the implicit suffix trees in $O(m)$ time. This sequence of implicit suffix trees may expose more information about S than does the single final suffix tree for S. Find a problem that can be solved more efficiently with the sequence of implicit suffix trees than with the single suffix tree. Note that the atgorithm cannot save the implicit suffix trees and hence the problem will have to be solved in parallel with the construction of the implicit suffix trees.

8. The naive Weiner algorithm for constructing the suffix tree of S (Section 6.2.1) can be described in terms of the Aho–Corasick algorithm of Section 3.4: Given string S of tength m, append \$ and let $\mathcal{P}$ be the set of patterns consisting of the $m + 1$ suffixes of string $S\$$. Then build a keyword tree for set $\mathcal{P}$ using the Aho–Corasick algorithm. Removing the backlinks gives the suffix tree for S. The time for this construction is $O(m^2)$. Yet, in our discussion of Aho–Corasick, that method was considered as a ***linear*** time method. Resolve this apparent contradiction.

9. Make explicit the relationship between link pointers in Weiner's algorithm and suffix links in Ukkonen's algorithm.

10. The time analyses of Ukkonen's algorithm and of Weiner's algorithm both rely on watching how the current node-depth changes, and the arguments are almost perfectly symmetric. Examine these two algorithms and arguments closely to make explicit the similarities and differences in the analysis. Is there some higher-level analysis that might establish the time bounds of both the algorithms at once?

11. Empirically evaluate different implementation choices for representing the branches out of the nodes and the vectors needed in Weiner's algorithm. Pay particular attention to the effect of alphabet size and string length, and consider both time and space issues in building the suffix tree and in using it afterwards.

12. By using implementation tricks similar to those used in Ukkonen's algorithm (particularly, suffix links and skip/count) give a linear-time implementation for McCreight's algorithm.

13. Flesh out the relationship between McCreight's algorithm and Ukkonen's algorithm, when they both are implemented in linear time.

14. Suppose one must dynamically maintain a suffix tree for a string that is growing or contracting. Discuss how to do this efficiently if the string is growing (contracting) on the left end, and how to do it if the string is growing (contracting) on the right end.

Can either Weiner's algorithm or Ukkonen's algorithm efficiently handle both changes to the right and to the left ends of the string? What would be wrong in reversing the string so that a change on the left end is "simulated by a change on the right end?

15. Consider the previous problem where the changes are in the interior of the string. If you cannot find an efficient solution to updating the suffix tree, explain what the technical issues are and why this seems like a difficult problem.

16. Consider a generalized suffix tree built for a set of k strings. Additional strings may be added to the set, or entire strings may be deleted from the set. This is the common case for maintaining a generalized suffix tree for biological sequence data [320]. Discuss the problem of maintaining the generalized suffix tree in this dynamic setting. Explain why this problem has a much easier solution than when arbitrary substrings represented in the suffix tree are deleted.